

CS422 - Programming Language Design

General Information and Introduction

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

General Information

- Class Webpage and Newsgroup:
<http://fsl.cs.uiuc.edu/~grosu> (go to “Classes”)
<news://class.cs422>
- Lectures: Tuesdays/Thursdays 3:30 - 4:45, 1302 Siebel Center
- Office hours: Mondays 10:00 - 12:00, 2110 Siebel Center
- Instructor: Grigore Roşu
 - Office: 2110 Siebel Center
 - Email: grosu@cs.uiuc.edu
 - WWW: <http://cs.uiuc.edu/grosu>
 - Secretary: Andrea Whitesell
(2106 SC, whitesel@uiuc.edu, 333-1043)
- Prerequisites: CS421 or equivalent, or instructor’s approval

- Textbooks

No textbook required! Self contained lecture notes will be posted on class' webpage. The following may be useful:

- 1) Friedman, Wand and Haynes, *Essentials of Programming Languages*, MIT Press, Second Edition, 2001
- 2) Winskel. *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993

- Other sources

- The Maude Language: <http://maude.cs.uiuc.edu>
- Proceedings of Conferences on Programming Languages
 - * **POPL** (ACM Symp. on Principles of Prog. Lang.)
 - * **PLDI** (ACM Symposium on Programming Language Design and Implementation)
 - * **OOPSLA** (ACM Conference on Object Oriented Programming Systems, Languages and Applications)

Grading

- Students registered for 4 units
 - Homework assignments (or MPs): **45%**
 - Final exam: **30%**
 - Individual project: **25%**
- Students registered for 3 units
 - Homework assignments: **60%**
 - Final exam: **40%**

The Homework Assignments

- This is a *labor intensive class*. The notions presented in class will be often backed by machine supported formalizations which you are supposed to modify or redo entirely as part of your assignments and as part of your project
- Assignments will be complete approximately every 4 lectures, on Tuesdays, and are due the next Tuesday.
- The exercises in the lecture notes are of two types: regular exercises and homework exercises:
 - *Exercise*. The regular exercises are intended to warm you up and to help you better understand other deeper notions.
 - *Homework Exercise*. The homework exercises are those which you have to solve as part of your homework.
- Therefore, you have *between 1 and 3 weeks for each exercise*.

The Unit Project

- The unit project will consist of designing a new programming language with specified features. This language will most likely be an extended version of an existing language. The design will be formalized and an interpreter will be provided, which we will test against many carefully selected programs.

The Final Exam

- The final exam will test your overall understanding of the concepts discussed in class, and it is expected to be consistent with your homework assignments' scores.

Collaboration and Other Policies

- You are free to discuss the homework assignments with other students (and are encouraged to do so!). The focus of any such discussion should be limited to figuring the problem specification, not coming up with a solution. *You may not jointly write or code any assignment.* To do so will be considered cheating! All cheating will be penalized by automatically assigning a failing grade for the course and instigating further disciplinary action with the appropriate university disciplinary body.
- You should retain copies of your assignments until you receive your final grade. In the event of a discrepancy between your scores on assignments and those on the exams, you may be asked to explain any work you performed. Your grade may be adversely affected by an inability to explain your work or by

failure to retain copies of it. All students are required to take the exam in order to receive a grade in the course.

- The course has a newsgroup. You are encouraged to use this group to ask questions, answer mundane system questions for other students, discuss homeworks, etc. In consideration for your peers, please don't use it to post flames, irrelevant messages, ads, etc.

Course Description

- Advanced course on principles of programming language design
- Major language design paradigms will be investigated and mathematically defined (or specified), including:
 - Static versus dynamic binding
 - Call-by-value, reference, name, need
 - Type checking and type inference
 - Objects and classes, concurrency
- Since the rigorous definitional framework will be *executable*, *interpreters for the designed languages will be obtained for free*
- Software analysis tools reasoning about programs in these languages will arise naturally
- Major theoretical models will be discussed

Tentative Subjects Covered in CS422

Operational semantics, equational semantics, Maude, inductive language definitions, several styles of defining and/or designing a simple programming language, defining a functional programming language, defining static and dynamic type checkers, type inference, defining typed and untyped object oriented languages, continuation-based definitions, continuation-passing style transformations, exceptions, concurrency, logic programming, denotational semantics, lambda-calculus, defining program analysers, etc.

Course Objectives

- Present and define rigorously the major features and design concepts in programming languages
- Show how elegantly and easily one can design a programming language if one uses the right tools and framework
- Understand the significant theory of programming languages
- The practical objective of this course is *not* to *implement* programming languages, but rather to *specify or define* them formally and modularly, on a feature by feature basis
 - Interpreters will, however, be obtained for free, because in the discussed framework one can *execute* specifications
 - For that reason, we take the freedom to interchangeably use the words *define* and *implement* in this course

Specification versus Implementation

What is the difference between *specification* and *implementation*?

- An intuitive way to think of them is to associate the first to the question **WHAT** and the second to the question **HOW**
- A specification says *what* properties a system should have, while an implementation says *how* those properties are achieved
- A specification declares the interface of a system as well as the operations that it can perform, together with properties that these must fulfill; e.g., “addition is commutative”. An implementation gives concrete implementations of these operations; it may require quite tricky algorithms.
- A specification can have many correct implementations. Such an implementation is said to *satisfy* the specification, and this is formally written $Impl \models Spec$

An Example: Integer Numbers

Integer numbers are part of any programming language that is worth its salt. What is the difference between *specifying* and *implementing* integer numbers?

- A *specification* of integer numbers may say
 - There are some entities called *integers*
 - There is some special integer called *zero* and written **0**
 - There are two unary operations *succ* and *pred*
 - There is some binary operation $+$
 - These operators have lots of properties, including:
 - * *succ* and *pred* are inverse to each other
 - * $+$ is associative, commutative and has **0** as identity
 - * $(\forall X, Y : integer) \text{succ}(X) + Y = \text{succ}(X + Y)$
 - * etc.

- A typical *implementation* of integer numbers may be based on a hardware binary representation. Let us consider for example one over $k + 1$ bits, known as *one's complement*: the most significant bit (the leftmost one) stays for the sign, 0 for positive and 1 for negative, and the remaining k bits stay for the absolute value of the integer number. Here, let's assume that overflows may lead to unpredictable results.

Exercise 1 *The one's complement binary implementation of integer numbers is **not** a correct implementation of integer numbers as specified above. Which properties are not satisfied?*

- In this course we will consider an idealistic implementation of integers, in which they can have any finite number of digits. The Maude 2.0 language that we will use in this class provides such an idealistic built-in representation of integer numbers

Specifying Programming Languages

- In this course the emphasis will be on *specifications of programming languages*. For a desired programming language, say \mathcal{PL} , the focus will be on devising a specification $Spec(\mathcal{PL})$, defining all the important aspects of \mathcal{PL} .
- Ideally, one would want to specify \mathcal{PL} on a feature by feature basis, in a modular way. Supposing that $\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_n$ are desired features of \mathcal{PL} , such as static binding, call-by-reference, etc., with specifications $Spec(\mathcal{F}_1), Spec(\mathcal{F}_2), \dots, Spec(\mathcal{F}_n)$, then a major goal in this class is to define, or rather “design”,

$$Spec(\mathcal{PL}) \text{ as } Compose(Spec(\mathcal{F}_1), Spec(\mathcal{F}_2), \dots, Spec(\mathcal{F}_n)),$$

where *Compose* is some appropriate module, or specification, composition operator.

Important Notes and Advice

- The lecture notes for this class will be all posted on the web and will be as detailed as needed. The other textbooks may be useful as auxiliary material in order to have a better understanding of the discussed concepts.
- We will *not* use Scheme, ML or OCAML in this class as implementation languages! These are quite advanced programming languages; using them to *implement* interpreters hides some of the real important and interesting issues in specifying a programming language.